MASSACHUSETTS INTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 89-558
September 1989

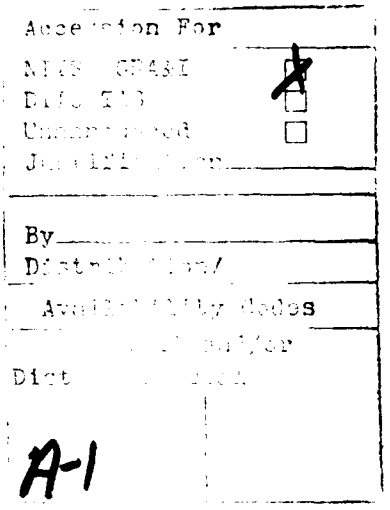
Optimum and Heuristic Algorithms for Finite State Machine Decomposition and Partitioning

Pravnav Ashar, Srinivas Devadas, and A. Richard Newton

Abstract

Techniques have been proposed in the past for various types of finite state machine (FSM) decomposition that use the number of states or edges in the decomposed circuits as the cost function to be optimized. These measures are not reflective of the true logic complexity of the decomposed circuits. These methods have been mainly heuristic in nature and offer limited guarantees as to the quality of the decomposition. In this paper we present optimum and heuristic algorithms for the general decomposition of FSMs such that *the sum total of the number of product terms in the one-hot coded and logic minimized submachines is minimum or minimal.* This cost function is much more reflective of the area of an optimally state-assigned and minimized submachine than the number of states/edges in the submachine. ~~We formulate~~ the problem of optimum two-way FSM decomposition ~~as~~ is one of *symbolic-output partitioning* and show that this is an easier problem than optimum state assignment. ~~We describe~~ a procedure of constrained prime-implicant generation and covering ~~that~~ represents an *optimum FSM decomposition algorithm, under the specified cost function.* Exact procedures are not viable for large problem instances. ~~We give~~ a novel iterative optimization strategy of *symbolic-implicant expansion and reduction*, modified from two-level Boolean minimizers, ~~that~~ represents a heuristic algorithm based on our exact procedure. Reduction and expansion are performed on functions with symbolic, rather than binary-valued outputs. ~~We present~~ preliminary experimental results ~~that~~ illustrate both the efficacy of the proposed algorithms and the validity of the selected cost function.

Microsystems Technology Laboratories | Massachusetts Institute of Technology | Cambridge Massachusetts 02139 | Room 39-321 Telephone (617) 253-0292



Acknowledgements

To be presented at the *International Conference of Computer Aided Design (ICCAD '89)* in November 1989. This work was supported in part by the Defense Advanced Research Projects Agency under contract number N00014-87-K-0825.

Author Information

Ashar and Newton: Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.
Devadas: Department of Electrical Engineering and Computer Science, Room 36-848, MIT, Cambridge, MA 02139. (617) 253-0454.

Optimum and Heuristic Algorithms for Finite State Machine Decomposition and Partitioning

Pranav Ashar Srinivas Devadas* A. Richard Newton

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract

Techniques have been proposed in the past for various types of finite state machine (FSM) decomposition that use the number of states or edges in the decomposed circuits as the cost function to be optimized. These measures are not reflective of the true logic complexity of the decomposed circuits. These methods have been mainly heuristic in nature and offer limited guarantees as to the quality of the decomposition. In this paper, we present optimum and heuristic algorithms for the general decomposition of FSMs such that *the sum total of the number of product terms in the one-hot coded and logic minimized submachines is minimum or minimal.* This cost function is much more reflective of the area of an optimally state-assigned and minimized submachine than the number of states/edges in the submachine. We formulate the problem of optimum two-way FSM decomposition as one of *symbolic-output partitioning* and show that this is an easier problem than optimum state assignment. We describe a procedure of constrained prime-implicant generation and covering that represents an *optimum FSM decomposition algorithm, under the specified cost function.* Exact procedures are not viable for large problem instances. We give a novel iterative optimization strategy of *symbolic-implicant expansion and reduction*, modified from two-level Boolean minimizers, that represents a heuristic algorithm based on our exact procedure. Reduction and expansion are performed on functions with symbolic, rather than binary-valued outputs. We present preliminary experimental results that illustrate both the efficacy of the proposed algorithms and the validity of the selected cost function.

1 Introduction

The area and performance optimization of sequential circuits is recognized as a key area. Work done in this area has involved the development of algorithms for state assignment (e.g. [6], [8]) and decomposition of finite state machines (e.g. [11], [10]).

Considerable progress has been made in the sequential logic synthesis arena in the recent past. Heuristic strategies for state assignment targeting two-level and multi-level logic implementations that achieve high-quality solutions have been developed (e.g. [8], [3]). State machine factorization algorithms have been developed and their relationships to the state assignment problem have been investigated in [4] [2].

In this paper, we address the problem of the decomposition of sequential machines into smaller interacting submachines, so as to optimize the area *and* performance of the resulting implementation. Previous approaches (e.g. [6], [4]) to finite state machine (FSM) decomposition have used the number of states and edges in the resulting submachines as their cost function. Given that the logic implementation of an FSM is derived from its State Transition Graph (STG) specification after state assignment and intensive logic optimization, this cost function does not reflect the true complexity of the eventual logic-level implementation and is, in fact, far from accurate. Previous approaches have been mainly heuristic in nature and offer limited guarantees as to the quality of the *final solution as well.*

The contributions of the work presented here include:

1. A formulation of the optimum two-way decomposition problem as one of *symbolic output partitioning*, with an associated cost function that is much closer to the final logic-level implementation than the number of states/edges in the decomposed submachines, namely, *the sum total of the number of product terms in the one-hot coded and logic minimized submachines.* This cost function allows us to *predict* the complicated effects of logic minimization.
2. The development of an exact solution to the above problem via a method of prime implicant generation and constrained covering. Exact methods for state assignment have been proposed in [5], but here we exploit the fact that the problem of two-way FSM decomposition is easier than that of state assignment. In particular, we present a polynomial-time algorithm to check for the validity of a given solution during prime implicant covering.
3. The development of a sophisticated heuristic optimization strategy that is applicable to problems of any size. We give a novel iterative optimization strategy of *symbolic implicant expansion and reduction*, modified from two-level Boolean minimizers, that represents a heuristic algorithm based on our exact procedure. Reduction and expansion are performed on functions with symbolic, rather than binary-valued outputs. Many different expansion/reduction heuristics have been implemented and evaluated under this global strategy.

We present basic definitions in Section 2. In Section 3, we formulate the decomposition problem as one of symbolic output partitioning and give an exact procedure to solve it. We give a theorem that proves the correctness of the procedure. A heuristic expand-reduce procedure, viable for large size problems, is presented in Section 4. Preliminary experimental results on area and performance optimization, that illustrate both the efficacy of the proposed algorithms, as well as the validity of our cost function, are presented in Section 5.

2 Preliminaries

A finite state machine is represented by its **State Transition Graph** (STG) or **State Transition Table** (STT), $G(V, E, W(E))$, where V is the set of vertices corresponding to the set of states S, where $\|S\|$ is the cardinality of the set of states of the FSM, an edge (v_i, v_j) joins v_i to v_j if there is a primary input that causes the FSM to evolve from state v_i to state v_j, and $W(E)$ is a set of labels attached to each edge, each label carrying the information of the value of the input that caused the transition and the values of the primary outputs corresponding to that transition.

A **partition** π on the set S is a collection of *disjoint* subsets whose set union is S. The disjoint subsets are called the **blocks** of π. A **factor** is N_R (≥ 1) sets of states and all fanout edges from these sets of states in the given machine. Each set of states is called an **occurrence** of the factor. The maximum number of states in any of the N_R occurrences of the the factor is denoted by N_F.

Given a State Graph description of a desired terminal behavior, the essence of the decomposition problem is to find two or more machines which, when interconnected in a prescribed way, will display that terminal behavior. We refer to the individual machines that make up the overall realization as **submachines**. The machine that results from the interconnection of the submachines is called the **decomposed machine**. By the **prototype machine**, we mean the machine that was used to define the terminal behavior to be realized.

3 Exact Procedure for 2-Way General Decomposition

General decompositions can have various topologies. We are concerned with the decomposition topology of Figure 1, where the original machine, M, has been decomposed into 2 submachines, M_1 and M_2, interconnected in the prescribed way. The output logic for the decomposed machine is distributed between the two submachines, unlike in [4] where a logic block external to the submachines was required to generate the primary outputs.

Optimal state assignment of a machine corresponds to finding an optimal multiple general decomposition of the machine. By multiple we mean that more than 2 submachines may be produced, interacting in much the same way as in Figure 1. The problem of a 2-way decomposition is thus simpler than the state assignment problem. Our goal is to provide exact or near-exact solutions to this problem.

3.1 Cost Function

The cost function for a general decomposition can vary depending on the eventual targeted implementation. Here, we are concerned with two-level implementations. The cost function used allows us to decompose the prototype machine into submachines such that the sum of the areas of the two-level implementation of each submachine after state assignment is less than or close to the area of the two-level implementation of the prototype machine after state assignment. The area of the two-level

*Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge

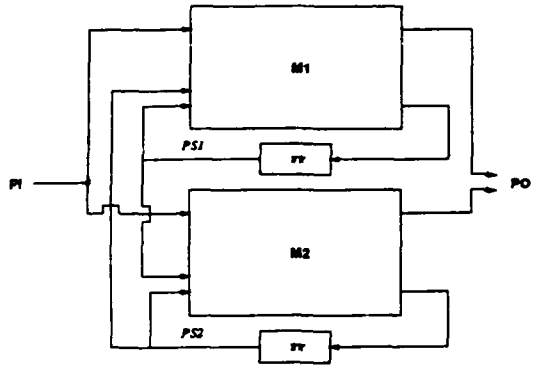


Figure 1: General Decomposition Topology

implementation of each submachine is *always* less than the area of the two-level implementation of the prototype machine. We also find that *the cost of the multi-level implementation of the decomposed machine obtained using this cost function is almost always less than the cost of* the multi-level implementation of the prototype machine. This implies that an optimal decomposition targeting a two-level implementation is a good decomposition for the multi-level case.

Consider the submachines in Figure 1. Let the number of product terms in the prototype machine, M, after one-hot coding and two-level Boolean minimization be P. Let the number of product terms in the submachines M_1 and M_2 after one-hot coding and two-level Boolean *minimization be* P_1 *and* P_2, respectively. We deem a decomposition to be optimum (optimal) if $P_1 + P_2$ is minimum (minimal). Note that in the case where no good decomposition can be found $P_1 + P_2 = P$. In this case, the best decomposition corresponds to a topological partition of the next state lines, which are produced by one-hot coding M (The next-state lines in a one-hot coded machine cannot share logic).

Since the two-level area of each submachine obtained using this cost function is always less than the two-level area of the prototype machine, the critical path of the decomposed machine in Figure 1 will be *smaller than the critical path of the prototype machine in the two-level* implementation. To optimize the critical path of the decomposed machine, the complexity of the prototype machine should be uniformly distributed between the submachines. A modified cost function of the form $P_1 + P_2 + \alpha||P_1 - P_2||$ characterizes the optimality of the decomposition with respect to timing also.

3.2 Decomposition, Factorization and Partitioning

We formulate the optimum decomposition problem in the sequel. We are given the initial State Transition Graph (STG) of M. Assume M has N states, $s_1, .. s_N$. We construct a function L as follows: The present-state (PS) field in the STG is replaced by an N-valued variable. The next-state (NS) field in M is *split* into *two* symbolic variables, i.e. s_i is split into symbolic outputs sa_1 and sb_1. s_2 is split into symbolic outputs sa_2 and sb_2 and so on. The primary input (PI) and output (PO) fields are untouched. An example transformation is shown below:

11 $s1$ $s2$ 10 — 11 100 $sa2$ $sb2$ 10

00 $s1$ $s3$ 01 — 00 100 $sa3$ $sb3$ 01

01 $s2$ $s2$ 11 — 01 010 $sa2$ $sb2$ 11

11 $s2$ $s3$ 00 — 11 010 $sb3$ $sb3$ 10

Consider the functions L_1 and L_2 with PI and PS fields that are the same as L. L_1 has the first NS sub-field, corresponding to the sa_i and the primary outputs. L_2 has the second NS sub-field corresponding to the sb_i. L_1 (left) and L_2 for our example are shown below:

11 100 $sa2$ 10 — 11 100 $sb2$

00 100 $sa3$ 01 — 00 100 $sb3$

01 010 $sa2$ 11 — 01 010 $sb2$

11 010 $sa3$ 00 — 11 010 $sb3$

L_1 and L_2 are topological partitions of the function L (which corresponds to the original STG), but they are also State Graphs of decomposed submachines, which together realize the behavior of M. To elaborate on this, we need to look at possible encodings of the sa_i and sb_i. Obviously, the codes for all the s_i have to be distinct. The codes for sa_k and sa_l can be the same if and only if the codes for sb_k and sb_l are different.

Assume a one-hot coding for the symbolic output of L_1 (L_2), with the extra degree of freedom that some of the sa_i (sb_i) can have the same code (Note that the present-state field has been replaced by a multiple-valued variable and has not been split as the next-state field has). That is, either the codes for sa_k and sa_l are the same or their bitwise intersection is all zeros. Let a one-hot code for L_1 (L_2) produce P_1 (P_2) product terms. P_1 can be changed only by making some of the sa_i the same, since a one-hot output coding is the worst case of no sharing between the eventual binary-valued outputs. If all the sa_i are coded with the same code, then we have merely the POs to realize and minimum cardinality of an encoded L_1 but the sb_i have all to be different. This will imply that $P_1 + P_2 \geq P$. If one coded all the sb_i to be the same, then L_2 is not required at all ($P_2 = 0$), but all the sa_i have to be coded differently and hence $P_1 = P$.

Thus, the problem is to decide which of sa_i can be coded the same under a one-hot code (implying that the corresponding sb_is are coded differently under a one-hot code), so as to produce a minimum $P_1 + P_2$. This is identical to identifying two *partitions* [6] in the original machine so that the one-hot coded decomposed machine corresponding to these partitions satisfies the cost function. When the subgraph associated with the states in one of the blocks of a partition has similar functionality to the subgraph associated with another block of states in the partition, this is exactly the same as identifying the best *factor* [1] in the original machine across states $s_1, .. s_N$, such that performing a one-hot coding on the factored and factoring machines separately using different fields, produces a minimum cumulative number of product terms. If sa_k is the same as sa_l, it means that s_k and s_l are both states in the same occurrence of the extracted factor. If sb_k is the same as sb_l, it means that s_k and s_l are (1) unselected states not in the factor or (2) correspondence states in different occurrences of the factor.

If one wishes to constrain the decomposition to extract factors with a maximum of N_F states in any occurrence of the factor, it implies that a maximum of N_F sa_i can be given the same code. If we require a factor with at least N_R occurrences it means that there have to exist at least N_R groups of sa_i such that the sa_i in each group have the same code.

3.3 Prime Implicant Generation and Covering

To solve an output encoding problem, we have to modify the prime implicant generation and covering strategies basic to Boolean minimization. We have a simpler (and slightly different) problem here from the classical output encoding problem, however, since we have assumed a one-hot coding and the only degree of freedom is in giving the same code to the symbolic outputs.

We have the functions L_1 and L_2 which have both binary-valued and a multiple-valued input, a symbolic output and binary-valued outputs (in the case of L_1). We generate generalized prime implicants (GPIs) for L_1 and L_2 much as in the Quine-McCluskey procedure with additional tags corresponding to the symbolic output. Initially, all minterms have tags corresponding to the symbolic output they assert. If a minterm that asserts a symbolic output sa_i, merges with a minterm asserting the symbolic output sa_l, the resulting cube has both tags sa_k and sa_l. A cube cancels another cube if and only if their tags are identical, their multiple-valued input parts are identical or the multiple-valued input part of the larger cube contains a one in all positions, and, the binary-valued input part of the larger cube covers the binary-valued input part of the smaller cube. Binary-valued outputs are treated the same as in the Quine-McCluskey procedure. When no larger cube can be generated, we have the set of all GPIs.

Given a set of GPIs for L_1 and L_2 one has to perform the selection of a cover such that $P_1 + P_2$ is minimum. The definition of a cover is different from classical minimization, since we have the constraint that all the s_i have to encoded differently. A cover has to contain all the minterms in L_1 (or L_2). In addition, given a set of GPIs for L_1, namely, G_1 and a set of GPIs G_2 for L_2, we have to check to see if we can code the sa_is and sb_is such that all the s_is have different codes and that no GPI has a multiple-valued input part that violates the encodeability constraints. Then, we can construct two submachines M_1 and M_2, which when one-hot coded would produce a cover cardinality of $|G_1|$ and $|G_2|$, respectively. The minimum covering problem corresponds to finding a minimum $|G_1| + |G_2|$. As mentioned earlier, trade-offs will exist.

It remains to clearly define how the constraint on distinct codes for the s_i affects the selection of G_1 and G_2. For this we need to inspect the symbolic tags of each of the implicants in G_1 and G_2, as well as their multiple-valued input parts. We can state the following:

1. If a prime implicant $p \in G_1$, has a tag containing sa_k, sa_l, sa_m, then it implies that sa_k, sa_l, sa_m have been given the same code.
2. If a prime implicant $p \in G_2$, has a tag containing sb_k, sb_l, sb_m, then it implies that sb_k, sb_l, sb_m have been given the same code.
3. If the multiple-valued input part of a prime implicant $p \in G_1$, G_2 has a 0 in the position corresponding to s_k and 1s in positions corresponding to s_l, s_m, then it implies that either the code for sa_k is different from the codes for *both* sa_l and sa_m or that the code for sb_k is different from the codes for *both* sb_l and sb_m.

3.4 Algorithm for Encodeability Check

Given the above relations, we have to check to see if all the s_i can have distinct codes for some selection of GPIs, G_1 and G_2. If so, the selection of G_1 and G_2 is valid. Else, the selection is invalid. The check can be accomplished in polynomial time, via the algorithm described below.

1. Construct a graph where each node is a state s_i and there is an edge with label a from s_i to s_j if they co-exist in the tag of some GPI in G_1. Similarly, there is an edge with label b from s_i to s_j if they co-exist in the tag of some GPI in G_2.
2. If any s_i, s_j pair has edges with both labels a and b, the selection is invalid, exit. We call this constraint the *uniqueness constraint*.
3. Since we attempt to identify partitions [6] in the prototype machine, we impose a transitivity constraint on the graph constructed in steps 1 and 2 above. This implies that if sa_i and sa_j have an edge with label a between them and sa_j and sa_k have an edge with label a between them, then sa_i and sa_k must also have an edge with label a between them. We define a *clique* as a subgraph such that each pair of constituent nodes is connected by edges with the *same* label. Thus, the constraint graph is composed of a set of cliques satisfying the following properties if the selection of GPIs does not violate step 2 above:
   - All the edges in a particular clique can have only one type of label. Thus, a clique can be identified with a label.
   - Two cliques with the same label cannot have a node in common unless both the cliques are contained in a single large clique.
   - Two cliques with a different label can have, at the most, one node in common. Thus, *any* two cliques can have, at the most, one node in common.
4. Once a graph that satisfies the encodeability constraints imposed by the output part of the GPIs is constructed, we check for violations of constraints imposed by the multiple-valued input part of the GPIs. Trivial input constraints are those with a 1 in all the positions of the multiple-valued input part or those with only one 1 in the multiple-valued input part (because s_i cannot have the same code as s_j for $i \neq j$ by virtue of steps 1 and 2). A selection of GPIs violates an input constraint *if and only if* there exists a multiple-valued input part in one of the selected GPIs and a pair of cliques in the constraint graph such that the following conditions are satisfied:
   - The intersection of the two cliques is non-null.
   - The intersection of the two cliques corresponds to an s_i such that the position associated with that s_i in the multiple-valued input part is a zero.
   - There is at least one s_i in *each* of the two cliques such that the position associated with that s_i in the multiple-valued input part is a one.
5. If no GPI exists such that its multiple-valued input part violates the constraints for any pair of cliques, the cover is deemed encodeable.

3.5 Correctness of the Exact Algorithm

Lemma 3.1 *The steps of the encodeability check algorithm are necessary and sufficient to ensure that the functionality of the decomposed machine is identical to that of the prototype machine.*

Proof: *Necessity*: The necessity of checking for the constraints in the algorithm follows from the description of the constraints in the previous sections.

Sufficiency: It can be shown that the functionality of the prototype machine is maintained if the satisfaction of the above-mentioned constraints is verified. Hence, no other constraints need to be checked for. ■

Lemma 3.2 *A minimum cardinality encodeable solution can be made up entirely of GPIs.*

Proof: Assume that we have a minimum cardinality solution with a cube c_1 that is not a GPI. We know that there exists a GPI covering c_1 that has the same tag as c_1, that its multiple-valued input part is either the same as that of c_1 or has a 1 in all its positions, that its binary-input part covers the binary-input part of c_1 and that its binary-output part covers the binary-output of c_1. Thus, replacing c_1 by the GPI does not change the functionality, the cardinality or the encodeability of the solution. Hence, a minimum cardinality solution can be made up entirely of GPIs. ■

Theorem 3.3 *The selection of a minimum cardinality encodeable cover for L_1 and L_2 from the GPIs represents an exact solution to the decomposition problem under the cost function being used.*

Proof: *The proof follows from Lemmas 3.1 and 3.2.* ■

Thus, we have an *exact* algorithm for solving the decomposition problem for the chosen cost-function. This algorithm can be extended to the problem of decomposition into multiple component machines. The reasons that this exact algorithm may not be viable for a given problem are that the number of GPIs may be too large and/or the covering problem may not be solvable in reasonable time. Therefore, we require a heuristic procedure to solve the problem.

4 Heuristic Procedure for 2-Way General Decomposition

The basic iterative strategy that has been used successfully for the two-level Boolean minimization problem appears promising for 2-way general decomposition also. The encodeability requirements for G_1 and G_2 are defined in the same manner as for the exact procedure. But, instead of enumerating all the GPIs, we begin with a set of GPIs for L_1 and L_2 and attempt to reduce their count, while maintaining the validity of the GPI covers. We can perform operations similar to the *reduce* and *expand* operations of MINI [7] and ESPRESSO [1,9] in an effort to minimize the cover cardinalities.

The three basic steps in our iterative loop are given below in the order in which they are carried out:

- Symbolic-reduce
- Symbolic-expand
- Minimize covers and remove input constraints

The cost function that we use for the iterative procedure is the same as that used for the exact algorithm, namely, the sum of the cardinalities of the minimized one-hot coded submachines. The steps in this loop are repeated until the cost of the solution, given by this cost function, remains unchanged after a pass through the loop.

In the symbolic-reduce and symbolic-expand steps, we attempt to modify the symbolic output tags of the GPIs that currently make up the covers G_1 and G_2 so that the possibility of obtaining a cover at the end of the *minimize* step, with a lower cardinality than the cardinality of the cover at the beginning of the current pass of the loop, is increased. The symbolic-reduce and symbolic-expand expand steps are analogous to the reduce and expand steps, respectively, in iterative two-level Boolean minimization. Unlike in two-level Boolean minimization, we do *not* check that the cost of the decomposition does not increase during the symbolic-reduce and symbolic-expand steps. On the other hand, the symbolic-reduce and symbolic-expand steps are carried out based on intelligent heuristics that do usually lead to a reduction in the cost *after* every pass through the loop. Even so, we maintain a copy of the best solution obtained up to the current point. The solution of the iterative procedure is the best solution obtained up to the point when the solution enters a local minimum that the iterative procedure cannot climb out of. The steps of the basic iterative loop are explained below.

In the **minimization** step, which follows the symbolic-expand, a two-level minimization of both the covers, G_1 and G_2, is carried out. This step incorporates *all* the cube-merging that becomes possible as a result of the symbolic-expand. We call the cover produced as a result of the minimization the *over-minimized* cover because the minimization is carried without taking into account violations of the input constraints, and may therefore not be encodeable. We unravel the multiple-valued input parts of the cubes to the minimum extent necessary to make the over-minimized cover encodeable. Whenever an input constraint violation is detected, the cardinality of the cover has to be increased by one to remove it. The procedure for detecting the input constraint violations and removing them is speeded up significantly by the use of intelligent pruning techniques that greatly reduce the search space.

The goal of the **symbolic-expand** procedure is to increase the size of the output tags of the GPIs in each cover till some form of primality is achieved. We consider the cover to be *prime* when no symbol can be added to the output tag of *any* GPI without violating the uniqueness constraint. The atomic operation in the expansion procedure inserts two states in the same output tag of a GPI, checking while doing so, that the uniqueness constraint is not violated. This atomic step can have two major effects:

- It makes new cube merges possible.
- It can result in additional input constraint violations.

Symbolic-expand is order dependent. The ordering heuristic attempts to maximize the occurrence of the first effect and minimize that of the second.

The **symbolic-reduce** operation transforms the prime cover into a non-prime cover. This operation is essential to the iterative process for moving out of the local minimum that it may have entered following the symbolic-expand and minimization steps. In converting a prime cover to a non-prime cover, the basic operation used by symbolic-reduce is to remove a state from the symbolic output tags that it is contained in, when the tags contain more than one state in them, while maintaining functionality. The states selected for removal are those whose insertion in the output tags of GPIs during the symbolic-expand generated new input constraints. Because the input constraints are generated due to non-null intersections between cliques with different labels, we ensure that after the symbolic-reduce, the intersection between *any* pair of cliques is null. Such a cover is said to be *maximally reduced*. This formulation of the symbolic-reduce operation is order independent.

The ordering of state pairs at the beginning of the symbolic-expand, which follows the symbolic-reduce operation, uses knowledge of the existence of cliques that remained after the symbolic-reduce. The new ordering attempts to explore a different direction for the symbolic-expand

Ex.	St.	Tot. I/P	Tot. O/P	2-level Area	Red. in Area (%)
dk16	27/2	8/8	8/1/9	1632/238/1870	20/88/8.1
ex1	5/4	13/9	22/2/24	2061/200/2264	15/92/7.1
fsl	16/5	15/5	5/3/8	3815/1386/5201	30/75/5.6
planet	32/2	13/13	14/11/25	3560/1776/5336	28/61/-17.6
s1a	8/3	13/13	3/2/5	1246/616/1862	45/73/17.7
scf	90/2	35/14	33/29/62	14832/3135/17967	20/83/2.8
scud	4/2	10/10	5/4/9	1050/792/1842	61/71/31.7
small[1]	5/3	7/5	5/2/7	259/113/372	40/74/13.3
styr	16/2	14/11	9/6/15	3996/1036/5032	5/75/-20.2
tbk	16/2	10/11	5/3/8	1275/425/1700	71/90/61.5

[1] Averaged over examples with area < 1500.
Whenever applicable, the first number in a box corresponds to submachine 1, the second to submachine 2 and the third is the overall number for the decomposed machine.

Table 1: Statistics of the Encoded Decomposed Machines

Ex.	Submach. 1[1]	Submach. 2[1]	Overall Area[1]
double	0.73	0.20	0.93
ex1	0.46	0.48	0.95
ex2	0.64	0.31	0.96
fsl	0.81	0.14	0.95
planet	0.67	0.30	0.97
s1a	0.53	0.21	0.74
scud	0.31	0.28	0.59
scf	0.52	0.51	1.03
styr	0.11	0.49	0.90
tbk	0.38	0.12	0.50

[1] Area normalized w.r.t. multi-level area of the prototype-machine
The misII-wolfe pipeline was used to compute the multi-level areas

Table 2: Comparision of the Multi-level Areas

that could lead to a better solution than the solution from the previous pass through the loop.

5 Results

We have implemented the heuristic procedure for optimal 2-way general decomposition in a program called **h-decom**. The input to **h-decom** is a KISS-style [8] State Table description of the prototype machine. As output, **h-decom** can produce either a fully encoded decomposed machine or a decomposed machine in which the PS inputs to the submachines and their NS outputs are symbolic. The efficacy of the decomposition can be based on a comparison of the following criteria:

- The areas of the two-level (multi-level) implementation of the encoded submachines and of the two-level (multi-level) implementation of the encoded prototype machine.
- The areas of the two-level implementation of the encoded submachines and of a vertically partitioned two-level implementation of the encoded prototype machine.
- The areas of the two-level (multi-level) implementation of the larger of the two encoded submachines and of the two-level (multi-level) implementation of the encoded prototype machine. The two-level area of the larger of the two submachines is an indicator of the critical path of the decomposed machine.
- The overall cardinalities of the two-level cover of the encoded and minimized decomposed submachines and of the two-level cover of the encoded and minimized prototype machine.
- The total number of inputs and outputs in each encoded submachine and in the encoded prototype machine.

The heuristic algorithm in **h-decom** was tested on a number of benchmark examples. The statistics of the examples are available in the public domain. A KISS-style encoding strategy was used on the submachines and the prototype machines. It can be observed from Table 1 that **h-decom** is successful in finding good 2-way general decompositions. It should be noted that the two-level area of each submachine is always substantially less than that of the prototype machine, implying that the *critical path of the decomposed circuit is always less than that for the prototype circuit*. It appears that the primary interest in using decomposition tools in industry stems from a need to improve the performance of FSM controllers, which often dictate the required duration of the system clock.

The total two-level area of the decomposed machine is usually less than that of the prototype machine. Since the two submachines have common inputs, some extra routing area is required, over and above the PLA area. However, this extra area is small in comparison to the PLA areas and does not offset the the area gain via decomposition. As can be seen from Table 1, the number of inputs for both submachines need not be equal. A submachine may be independent of some of the primary inputs and present state lines.

It is apparent from the topology of Figure 1 that we do not add extra levels of logic to the network. Thus, the reduction in area is as a result of the same causes as in the vertical partitioning of PLAs. In general, it is not necessary that a good vertical partition should exist for a PLA. Our method of decomposition ensures that a good vertical partition does exist, which implies that the performance of the FSM can be improved without compromising the area.

We also report the multi-level areas for the large examples, for which a two-level implementation may not be efficient, in Table 2. It can be seen that the multi-level area of the decomposed machine is *almost always* smaller than that of the prototype machine, even though our decomposition strategy is not geared specifically toward decomposition for optimizing multi-level area. To obtain the multi-level areas, decompositions targeting optimal performance were used as starting points. The results imply that a good decomposition targeting two-level area is usually a good decomposition for the multi-level case.

These results are significantly better than those obtained via factorization [4].

6 Conclusions

We have proposed exact and heuristic algorithms for optimum and optimal 2-way general decomposition of finite state machines. These algorithms are based on a cost function that is more reflective of the cost of the logic-level implementation of the decomposed machine than the cost function used by previous approaches to the decomposition problem. We have implemented the heuristic algorithm in the program **h-decom**. Good decompositions were obtained using **h-decom** for a large number of benchmark examples.

7 Acknowledgements

The interesting discussions with Tony Ma and Wayne Wolf on sequential logic synthesis problems are acknowledged. This research was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-87-K-0825 and N00039-87-C-0182, Digital Equipment Corporation, AT&T Bell Laboratories and Semiconductor Research Corporation. Their support is gratefully acknowledged.

References

[1] R. K. Brayton, G. D. Hachtel, Curt McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[2] S. Devadas. General decomposition of sequential machines: relationships to state assignment. In *Proc. of 26th Design Automation Conference*, pages 314-320, June 1989.

[3] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Mustang: state assignment of finite state machines targeting multi-level logic implementations. In *IEEE Transactions on CAD*, pages 1290-1300, December 1988.

[4] S. Devadas and A. R. Newton. Decomposition and factorization of sequential finite state machines. In *Int'l Conference on Computer-Aided Design*, November 1988.

[5] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment and four-level boolean minimization. In *Electronics Research Laboratory Memorandum M89/8*, University of California, Berkeley, February 1989.

[6] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

[7] S. J. Hong, R. G. Cain, and D. L. Ostapko. Mini: a heuristic approach for logic minimization. In *IBM journal of Research and Development*, pages 443-458, September 1974.

[8] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment of finite state machines. In *IEEE Transactions on CAD*, pages 269-285, July 1985.

[9] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for pla optimization. In *Proc. IEEE Int. Conf. on CAD (ICCAD)*, pages 352-355, 1986.

[10] M. Yoeli. The cascade decomposition of sequential machines. In *IRE Trans. Electronic Computers*, pages 587-592, April 1961.

[11] H. P. Zeiger. Loop-free synthesis of finite-state machines. In *MIT Ph.D Thesis*, Department of Electrical Engineering, Cambridge, Mass., September 1964.